

# P&S Heterogeneous Systems

## Parallel Patterns: Reduction

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2022

7 November 2022

# Performance Considerations

# Traditional Program Structure

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU

Serial Code (host)

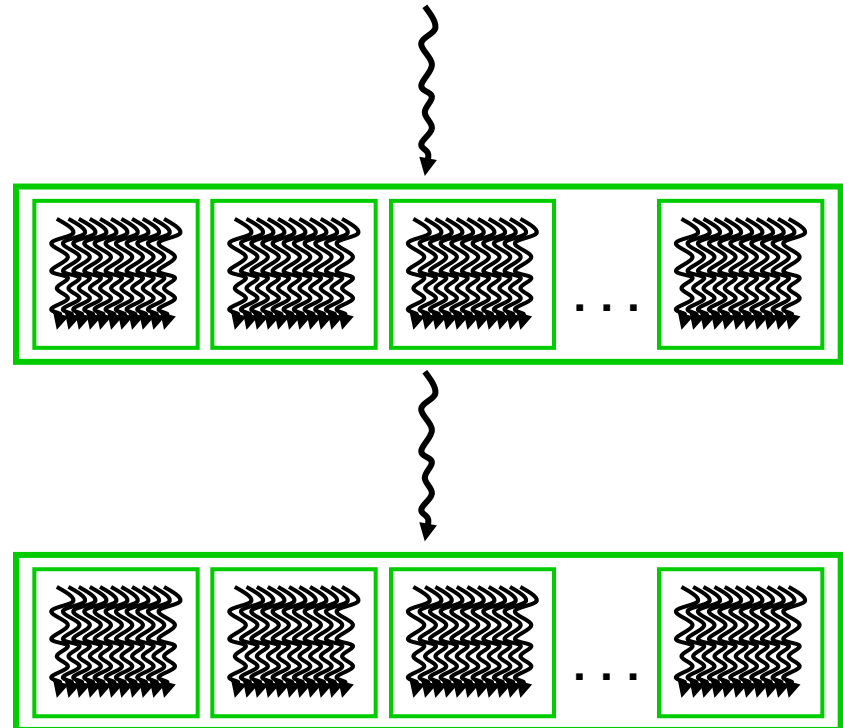
Parallel Kernel (device)

```
KernelA<<< nBlk, nThr >>>(args);
```

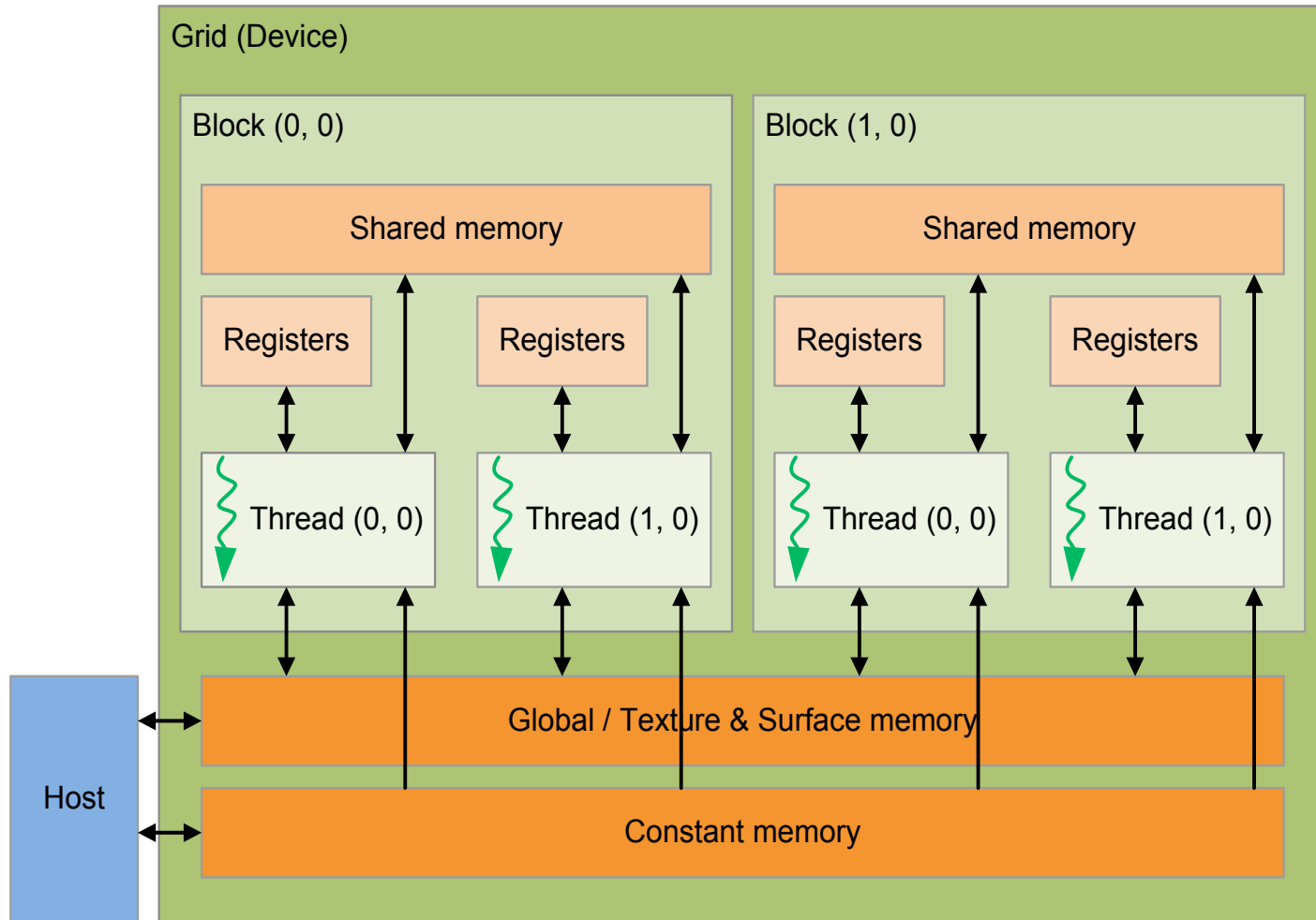
Serial Code (host)

Parallel Kernel (device)

```
KernelB<<< nBlk, nThr >>>(args);
```

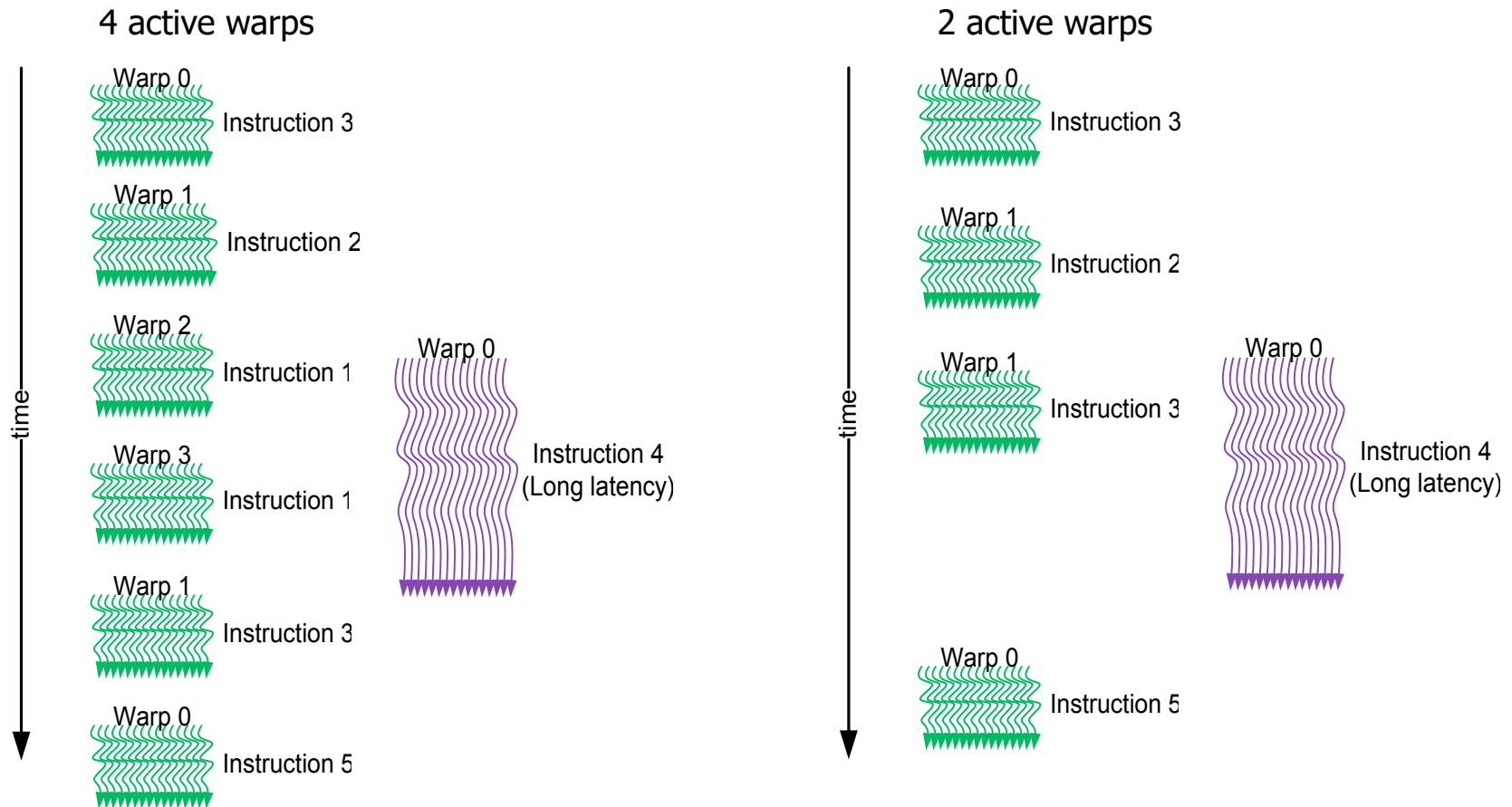


# Memory Hierarchy in CUDA Programs



# Latency Hiding and Occupancy

- FGMT can hide **long latency operations** (e.g., memory accesses)
- **Occupancy**: ratio of **active warps** to the maximum number of warps per GPU core



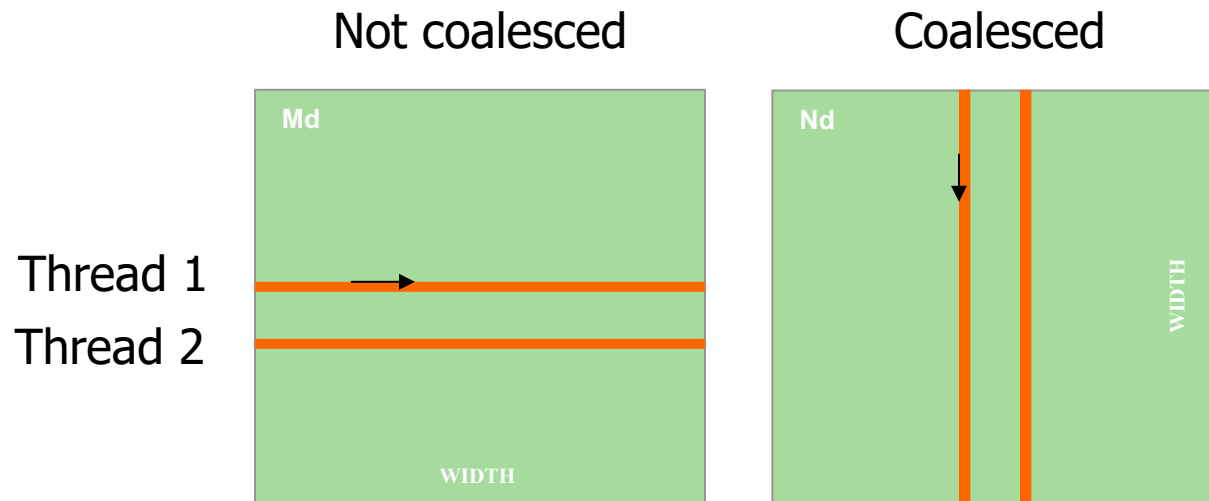
# Memory Coalescing (I)

---

- When threads in the same warp access consecutive memory locations in the same burst, the accesses can be combined and served by one burst
  - One DRAM transaction is needed
  - Known as memory coalescing
- If threads in the same warp access locations not in the same burst, accesses cannot be combined
  - Multiple transactions are needed
  - Takes longer to service data to the warp
  - Sometimes called memory divergence

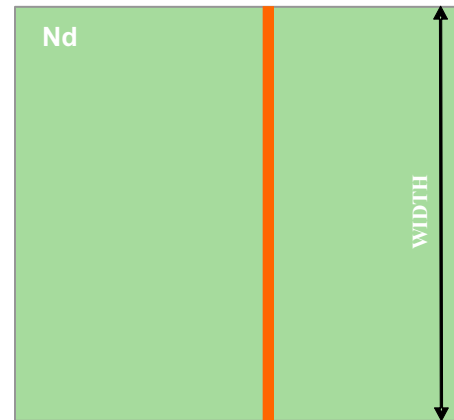
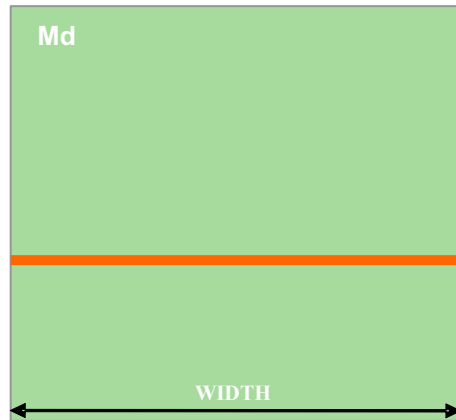
# Memory Coalescing (II)

- When accessing global memory, we want to make sure that **concurrent threads access nearby memory locations**
- **Peak bandwidth** utilization occurs when all threads in a warp access **one cache line** (or several consecutive cache lines)

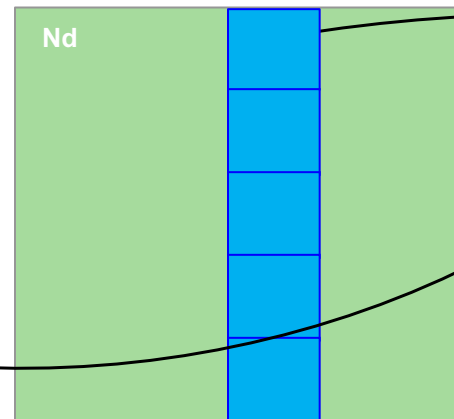
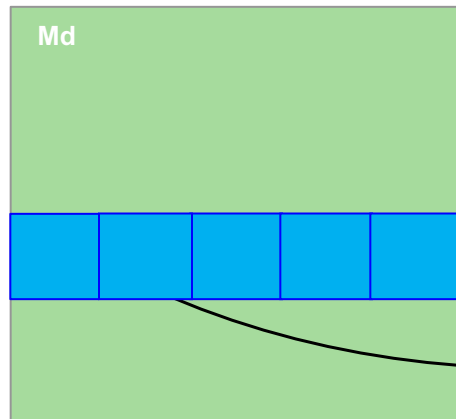


# Use Shared Memory to Improve Coalescing

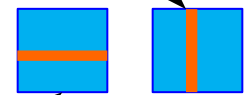
Original  
Access  
Pattern



Tiled  
Access  
Pattern



Copy into  
scratchpad  
memory



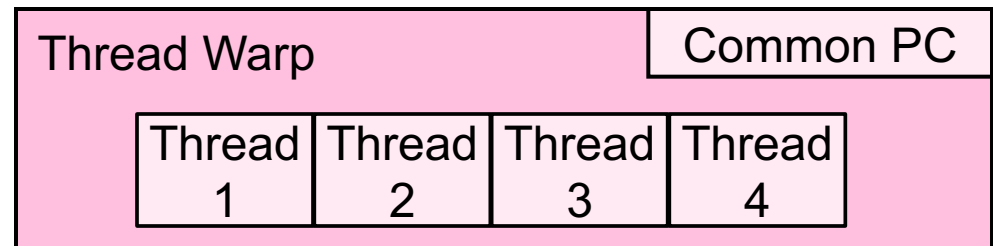
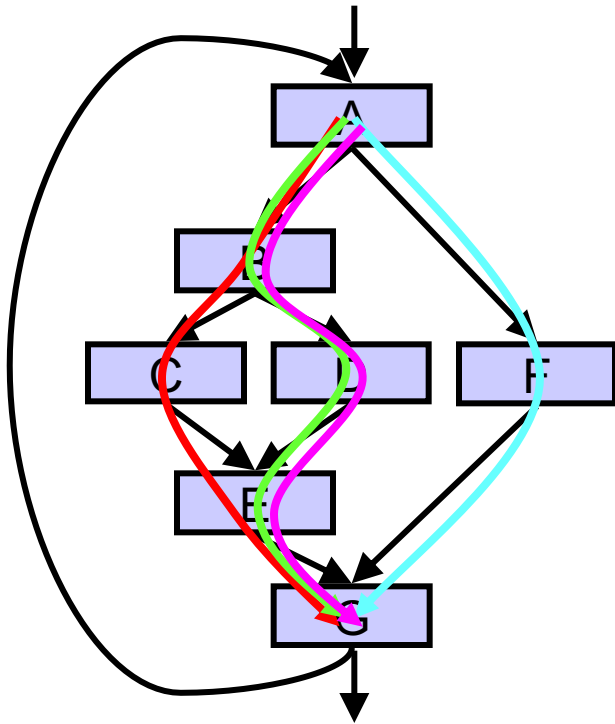
Perform  
multiplication  
with scratchpad  
values



# SIMD Utilization

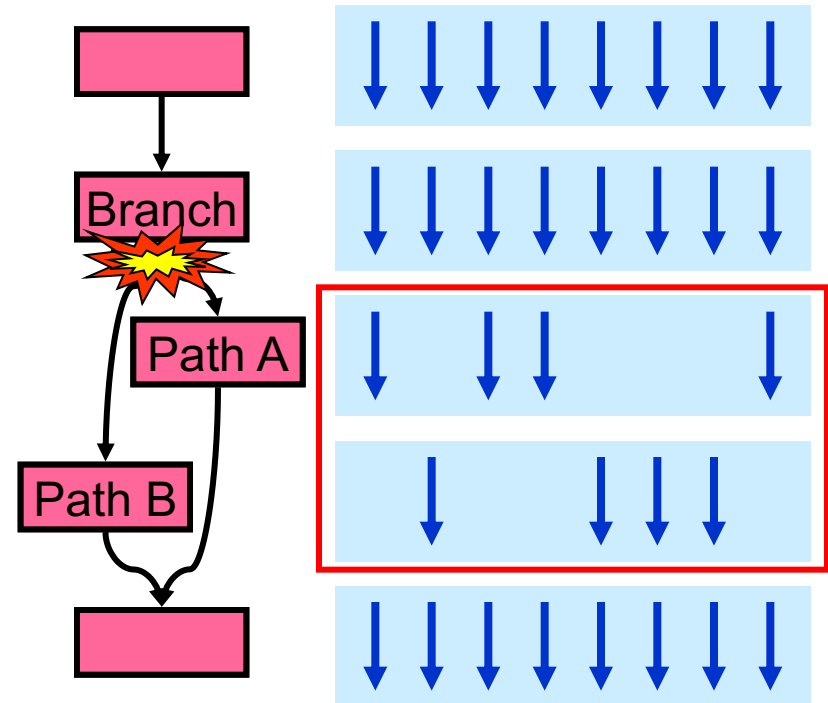
# Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have **conditional control flow instructions**
- Threads can execute different control flow paths



# Control Flow Problem in GPUs/SIMT

- A GPU uses a SIMD pipeline to save area on control logic
  - Groups scalar threads into warps
- **Branch divergence** occurs when threads inside warps branch to different execution paths

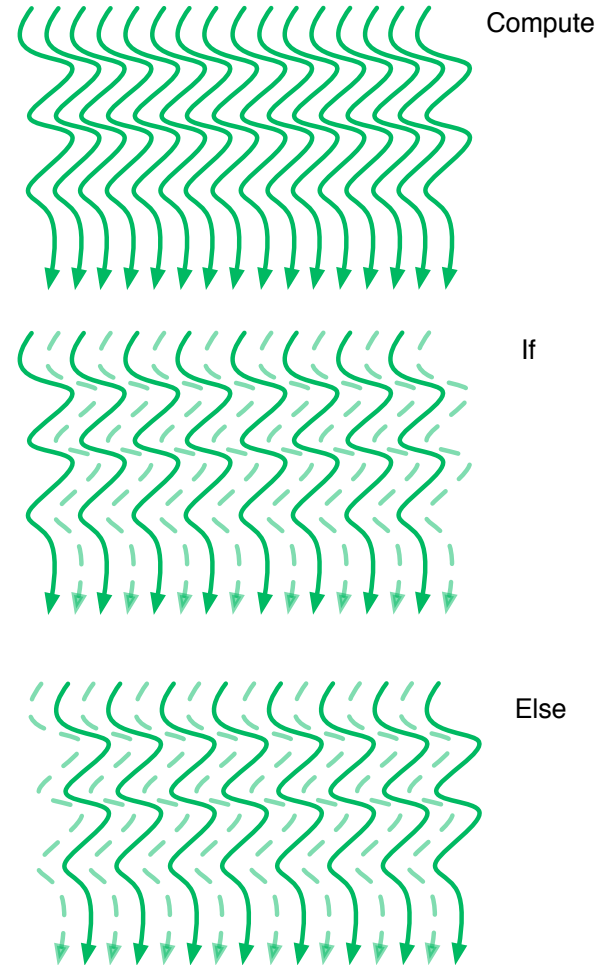


**This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations?**

# SIMD Utilization

## ■ Intra-warp divergence

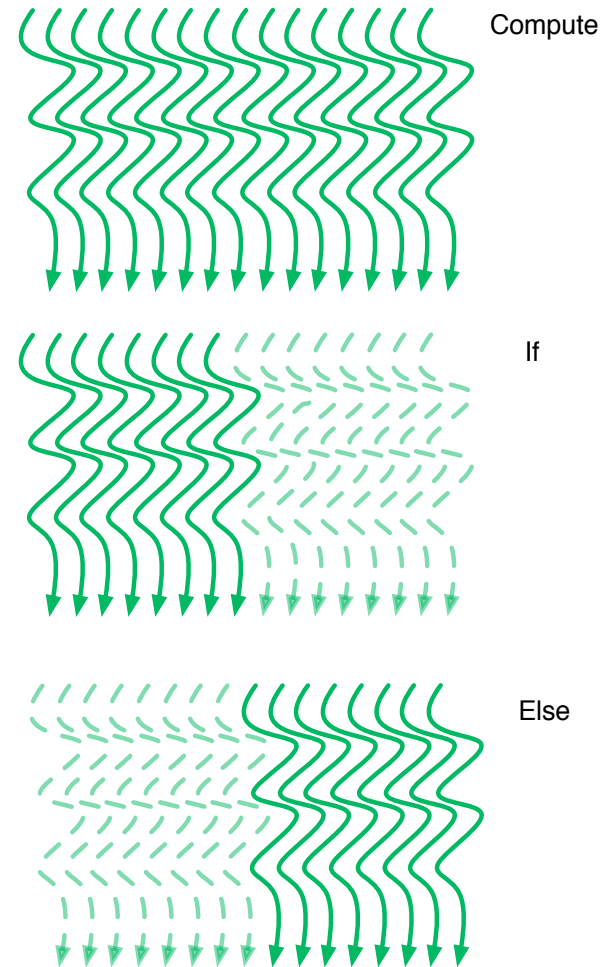
```
Compute(threadIdx.x);  
if (threadIdx.x % 2 == 0){  
    Do_this(threadIdx.x);  
}  
else{  
    Do_that(threadIdx.x);  
}
```



# Increasing SIMD Utilization

## ■ Divergence-free execution

```
Compute(threadIdx.x);  
if (threadIdx.x < 32){  
    Do_this(threadIdx.x * 2);  
}  
else{  
    Do_that((threadIdx.x%32)*2+1);  
}
```



# Reduction Operation

# Reduction Operation

---

- A **reduction** operation reduces a set of values to a single value
  - Sum, Product, Minimum, Maximum are examples
- **Properties of reduction**
  - Associativity
  - Commutativity
  - Identity value
- Reduction is a key primitive for parallel computing
  - E.g., MapReduce programming model

# Sequential Reduction

---

- A sequential implementation of reduction only needs a `for` loop to go through the whole input array
  - $N$  elements  $\rightarrow$   $N$  iterations



```
sum = 0; // Initialize with identity value
```

```
for(i = 0; i < N; ++i) {
```

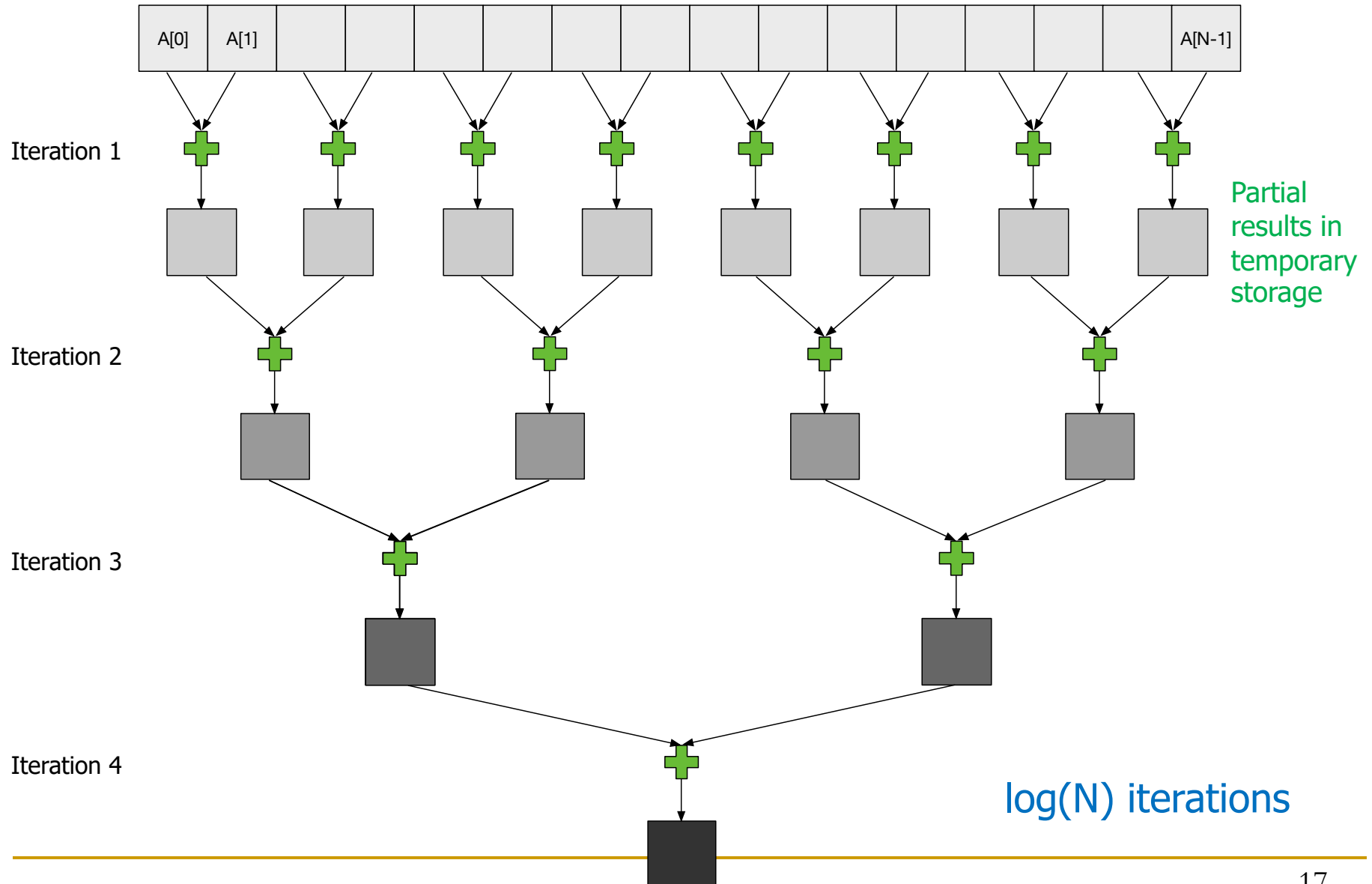
```
    sum += A[i]; // Accumulate elements of input array A[]
```

```
}
```

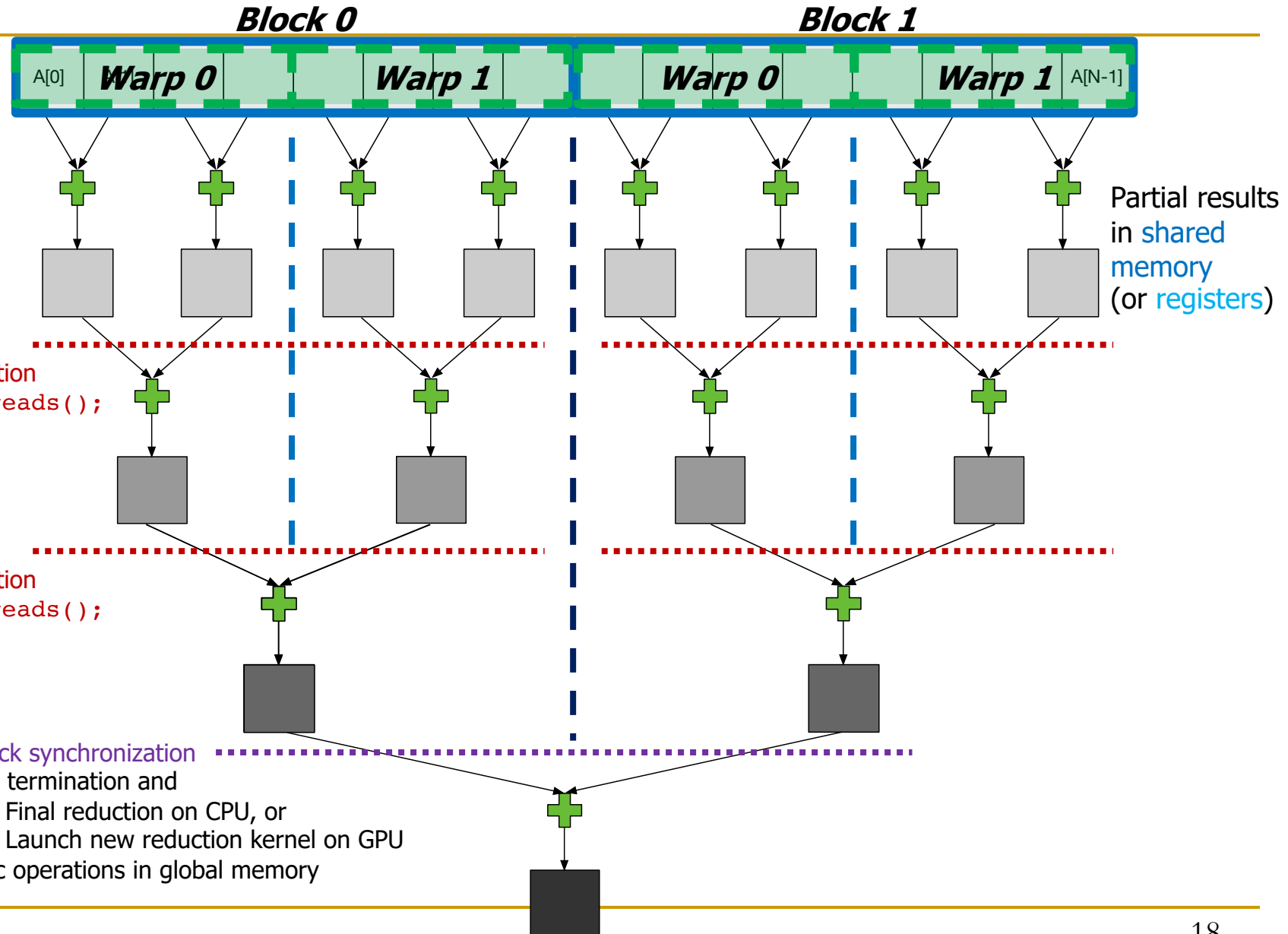
- Many independent operations
  - A parallel implementation can calculate multiple partial sums, and then reduce them



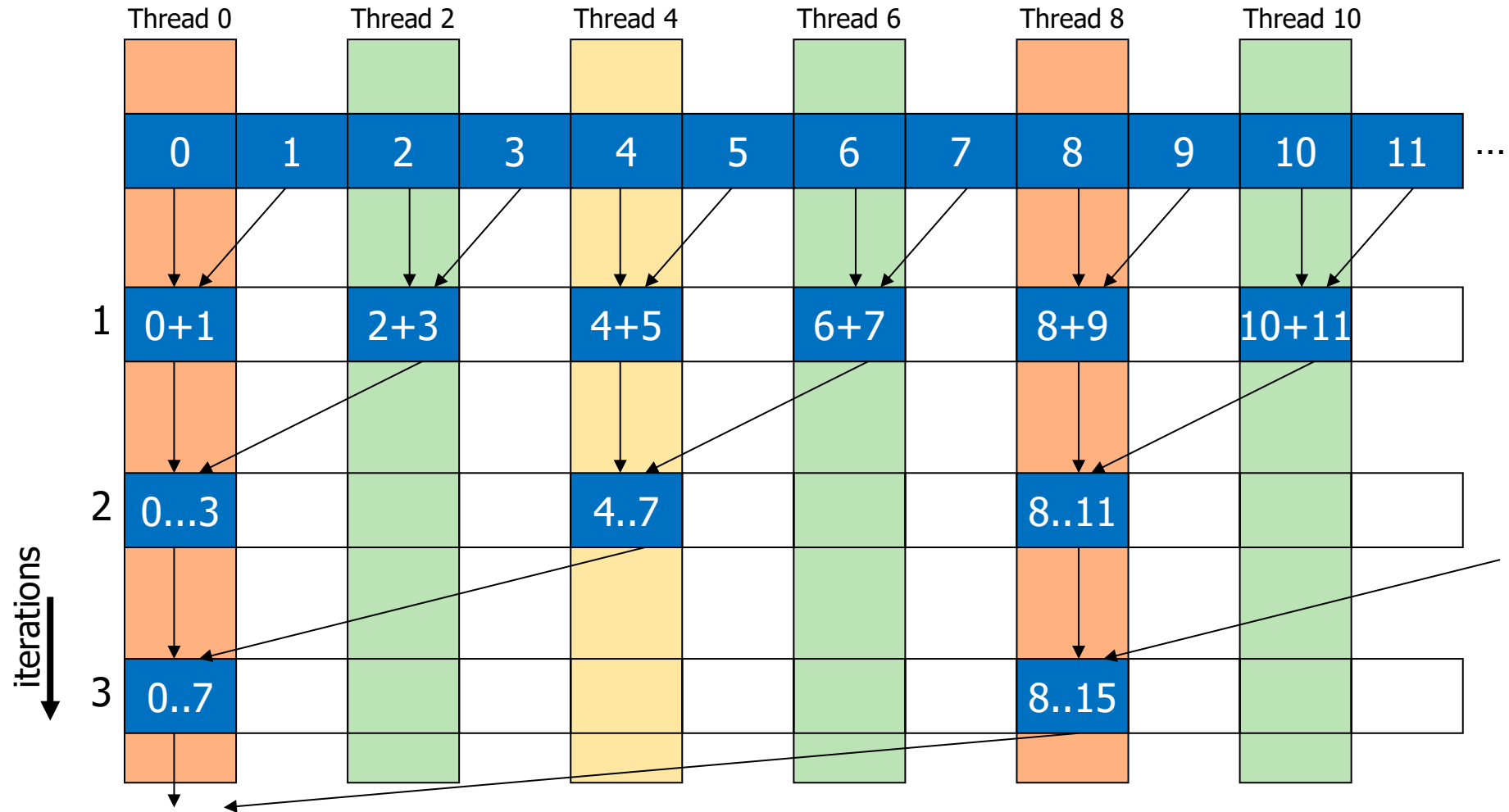
# Tree-Based Reduction



# Tree-Based Reduction on GPU



# Vector Reduction: Naïve Mapping (I)

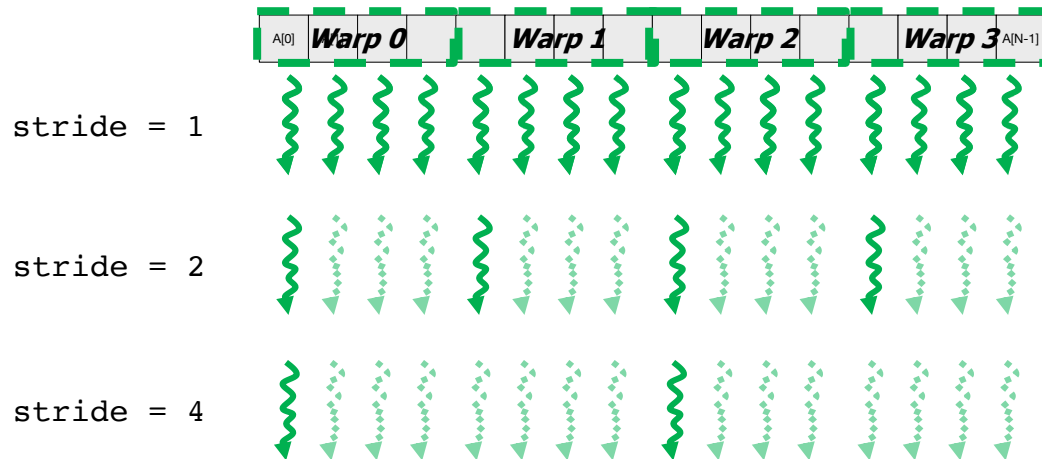


# Vector Reduction: Naïve Mapping (II)

## ■ Program with **low SIMD utilization**

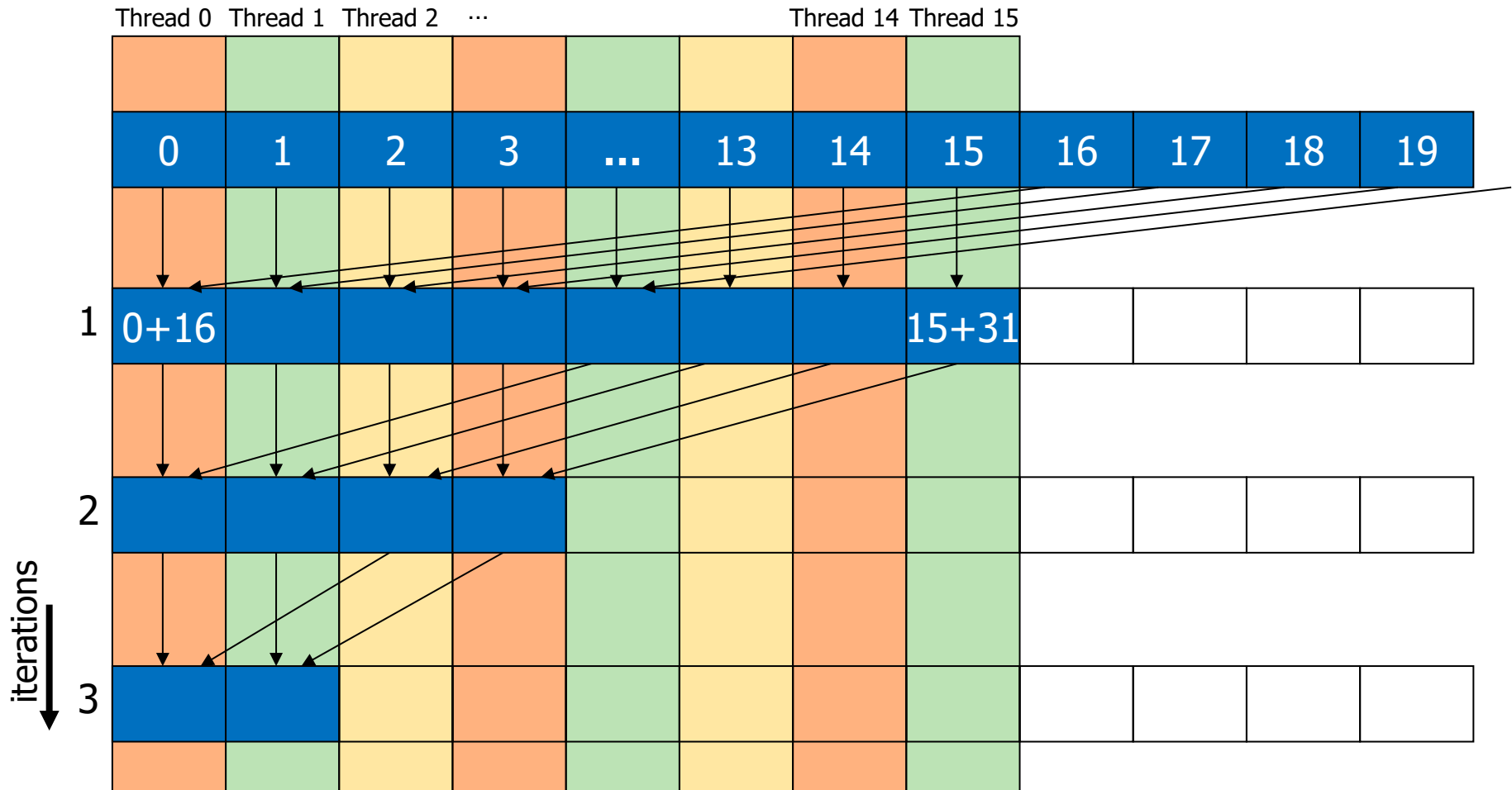
```
__shared__ float partialSum[]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = 1; stride < blockDim.x; stride *= 2){  
  
    __syncthreads();  
  
    if (t % (2*stride) == 0)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

How to avoid the  
**warp underutilization?**



# Divergence-Free Mapping (I)

- All active threads belong to the same warp

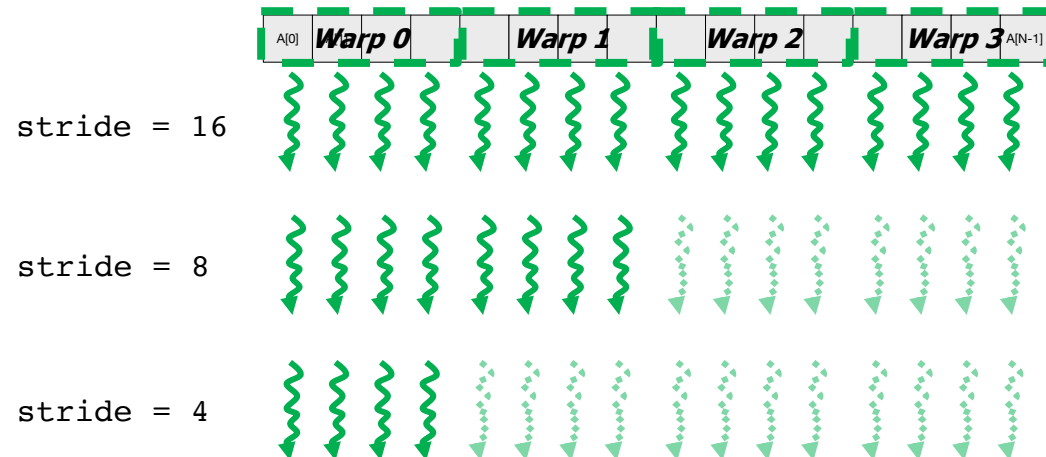


# Divergence-Free Mapping (II)

## ■ Program with high SIMD utilization

```
__shared__ float partialSum[]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = blockDim.x; stride > 0;  stride >> 1){  
  
    __syncthreads();  
  
    if (t < stride)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

Warp utilization  
is maximized

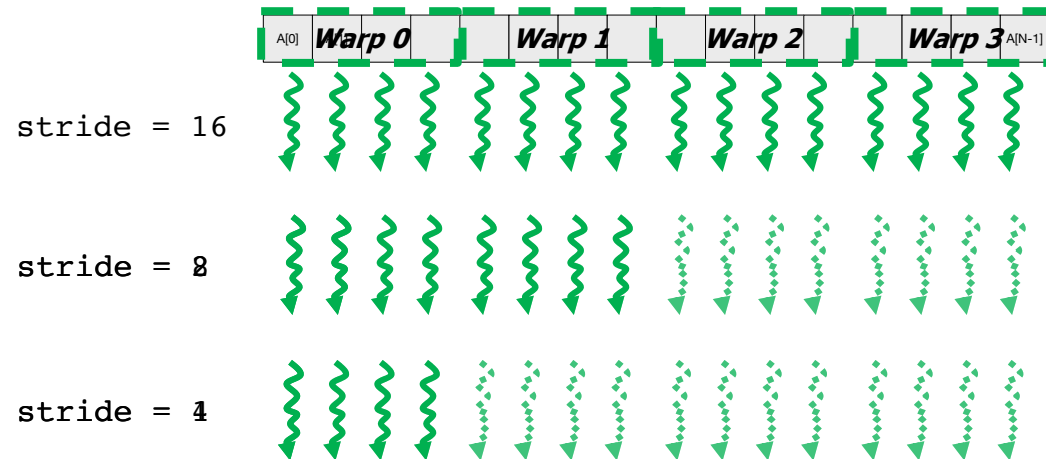


# Divergence-Free Mapping (III)

## ■ Program with high SIMD utilization

```
__shared__ float partialSum[]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = blockDim.x; stride > 0;  stride >> 1){  
  
    __syncthreads();  
  
    if (t < stride)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

We can use **warp shuffle**  
to avoid  
shared memory accesses  
and `__syncthreads()`



# Warp Shuffle Functions

---

- Built-in **warp shuffle functions** enable threads to share data with other threads in the same warp
  - Faster than using shared memory and `__syncthreads()` to share across threads in the same block
- Variants:
  - `__shfl_sync(mask, var, srcLane)`
    - Direct copy from indexed lane
  - `__shfl_up_sync(mask, var, delta)`
    - Copy from a lane with lower ID relative to caller
  - `__shfl_down_sync(mask, var, delta)`
    - Copy from a lane with higher ID relative to caller
  - `__shfl_xor_sync(mask, var, laneMask)`
    - Copy from a lane based on bitwise XOR of own lane ID



# Read and Write Access to GPU Shared Memory

- Threads running on **processing engines** have access to a local **register file** (LRF)
- And **shared memory** banks (SRF)

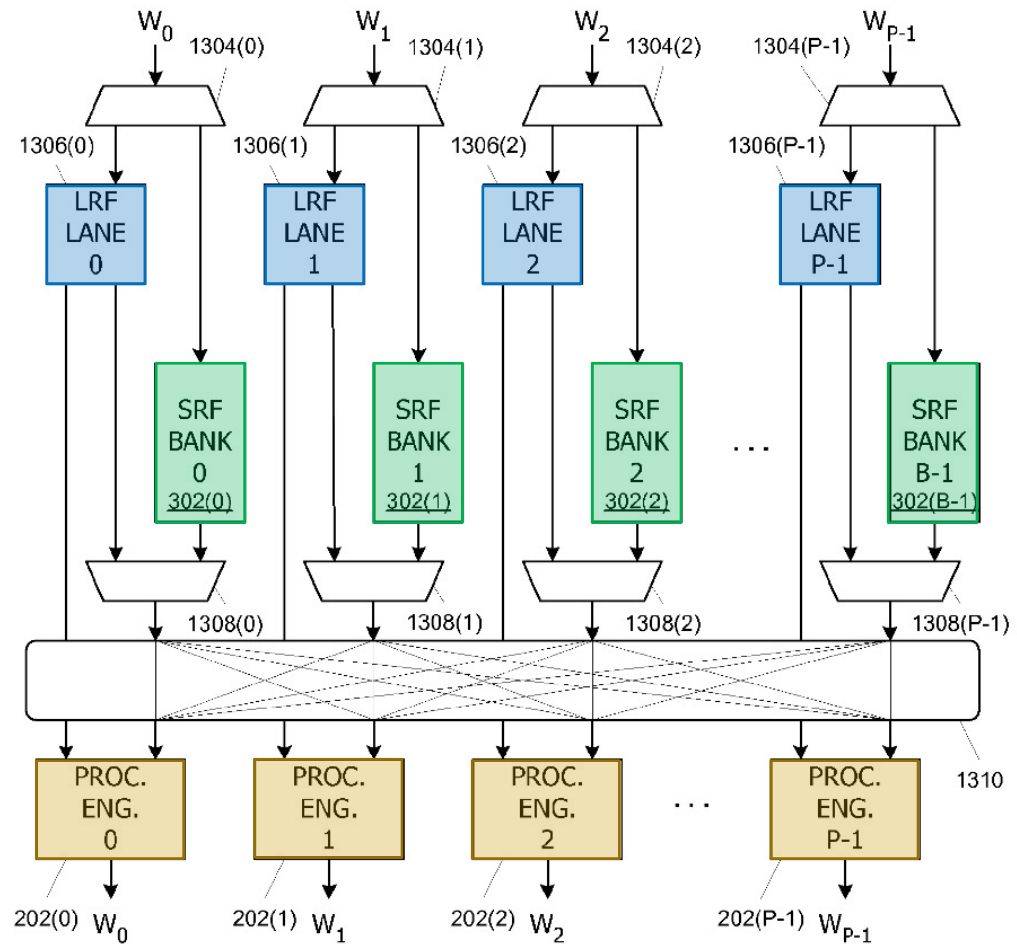
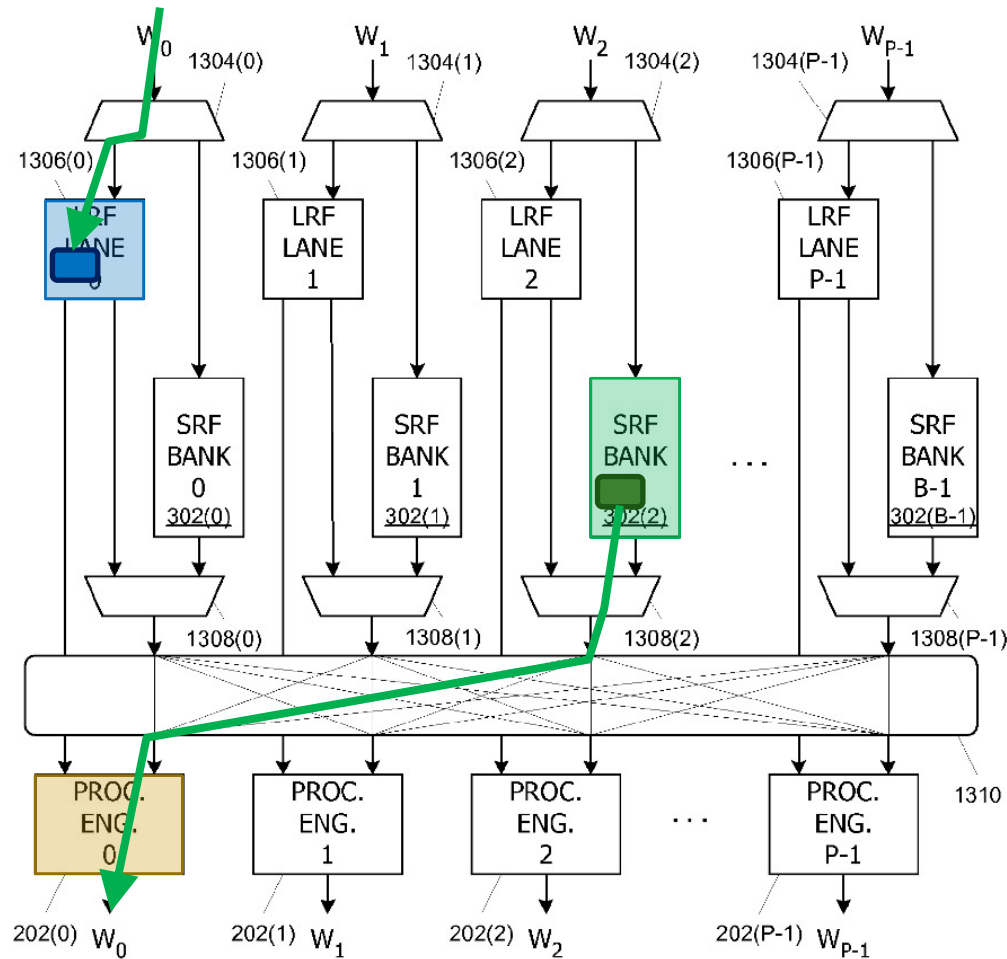


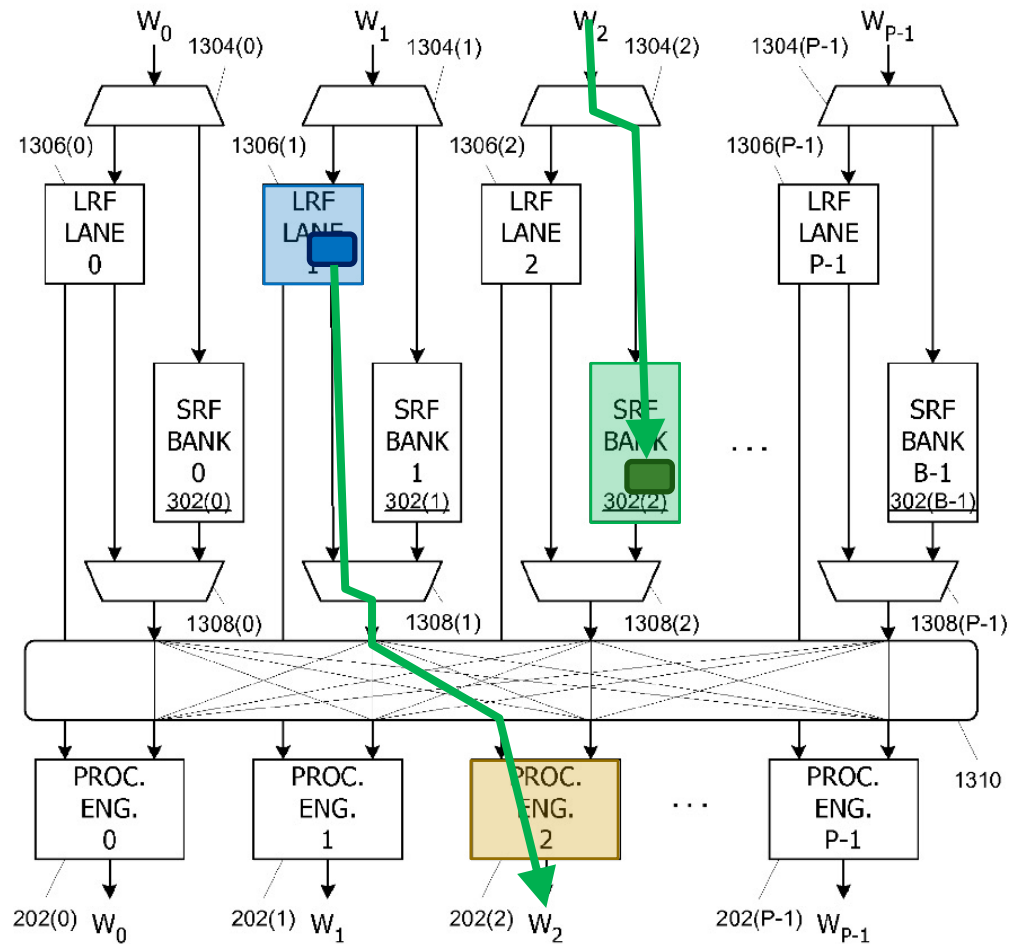
FIG. 13

# Read from Shared Memory Bank



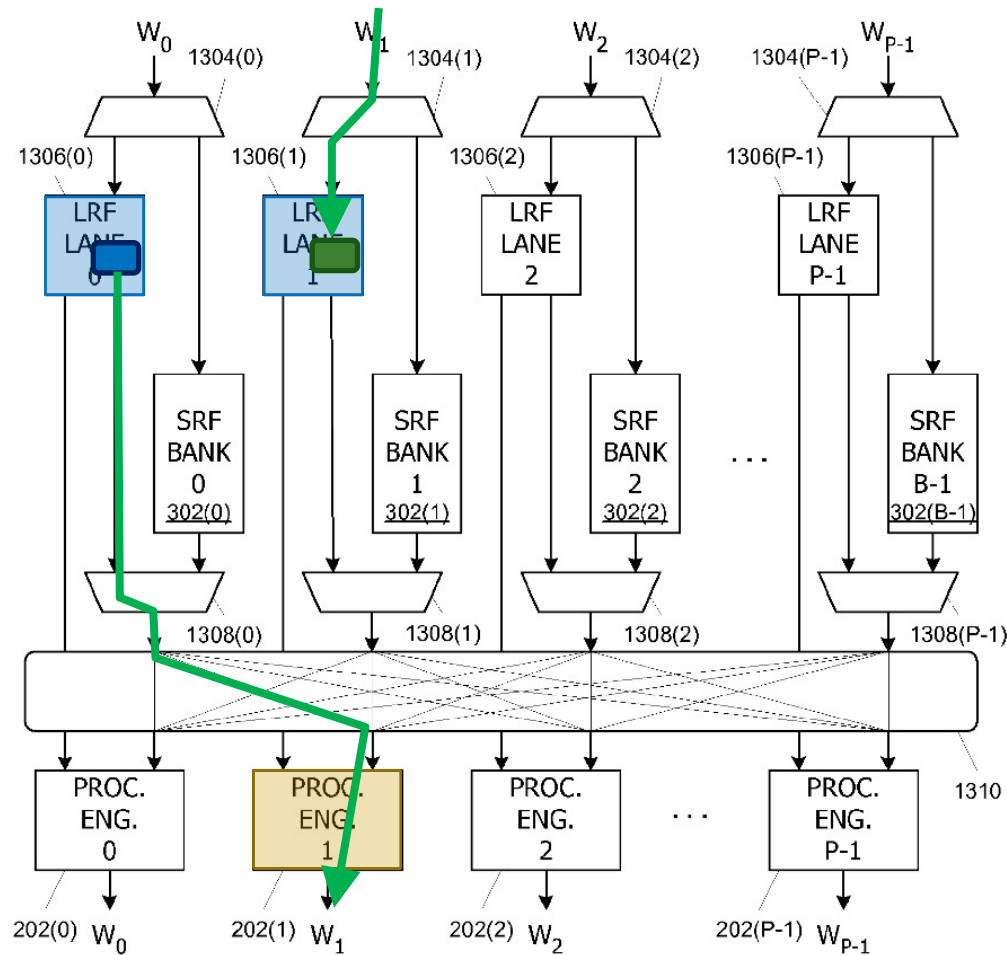
*FIG. 13*

# Write to Shared Memory Bank



*FIG. 13*

# Shuffling Operations within a Warp



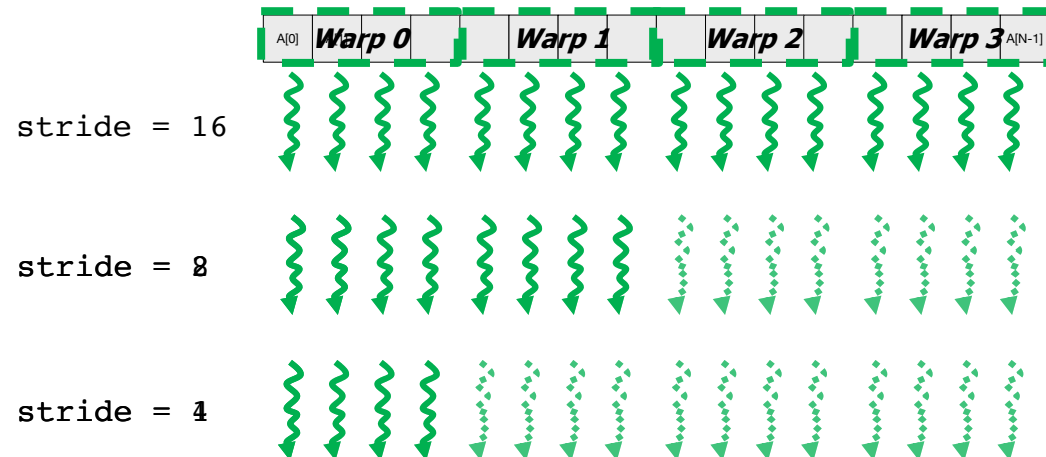
**FIG. 13**

# Divergence-Free Mapping (III)

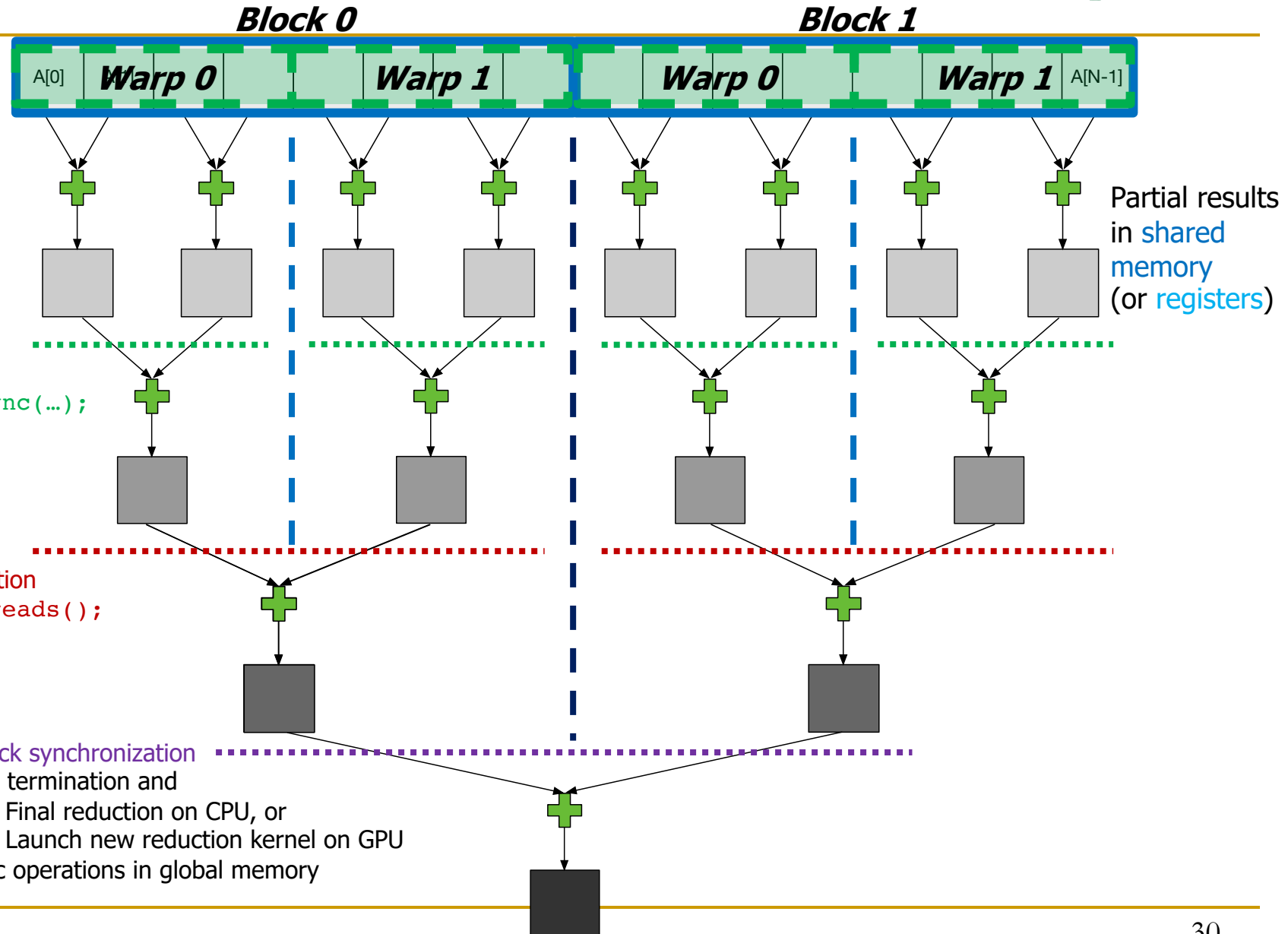
## ■ Program with high SIMD utilization

```
__shared__ float partialSum[]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = blockDim.x; stride > 0;  stride >> 1){  
  
    __syncthreads();  
  
    if (t < stride)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

We can use **warp shuffle**  
to avoid  
shared memory accesses  
and `__syncthreads()`



# Tree-Based Reduction on GPU (with Warp Shuffle)



# Reduction with Warp Shuffle

```
__global__ void reduce_kernel(float* input, float* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ float input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();


    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction tree with shuffle instructions
    float sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];

        for(unsigned int stride = WARP_SIZE/2; stride > 0; stride /= 2) {
            sum += __shfl_down_sync(0xffffffff, sum, stride);
        }
    }
    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
```

# Warp Reduce Functions

- Ampere (cc 8.x) adds native support for warp-wide reduction operations

 **DEVELOPER ZONE** **CUDA TOOLKIT DOCUMENTATION**

- ▶ B.1. Function Execution Space Specifiers
- ▶ B.2. Variable Memory Space Specifiers
- ▶ B.3. Built-in Vector Types
- ▶ B.4. Built-in Variables
- B.5. Memory Fence Functions
- B.6. Synchronization Functions
- B.7. Mathematical Functions
- ▶ B.8. Texture Functions
- ▶ B.9. Surface Functions
- B.10. Read-Only Data Cache Load Function
- B.11. Load Functions Using Cache Hints
- B.12. Store Functions Using Cache Hints
- B.13. Time Function
- ▶ B.14. Atomic Functions
- ▶ B.15. Address Space Predicate Functions
- ▶ B.16. Address Space Conversion Functions
- ▶ B.17. Alloca Function
- ▶ B.18. Compiler Optimization Hint Functions
- B.19. Warp Vote Functions
- ▶ B.20. Warp Match Functions
- ▼ **B.21. Warp Reduce Functions**
  - B.21.1. Synopsis
  - B.21.2. Description

## B.21. Warp Reduce Functions

The `__reduce_sync(unsigned mask, T value)` intrinsics perform a reduction operation on the data provided in `value` after synchronizing threads named in `mask`. `T` can be unsigned or signed for {add, min, max} and unsigned only for {and, or, xor} operations.

Supported by devices of compute capability 8.x or higher.

### B.21.1. Synopsis

```
// add/min/max
unsigned __reduce_add_sync(unsigned mask, unsigned value);
unsigned __reduce_min_sync(unsigned mask, unsigned value);
unsigned __reduce_max_sync(unsigned mask, unsigned value);
int __reduce_add_sync(unsigned mask, int value);
int __reduce_min_sync(unsigned mask, int value);
int __reduce_max_sync(unsigned mask, int value);

// and/or/xor
unsigned __reduce_and_sync(unsigned mask, unsigned value);
unsigned __reduce_or_sync(unsigned mask, unsigned value);
unsigned __reduce_xor_sync(unsigned mask, unsigned value);
```

### B.21.2. Description

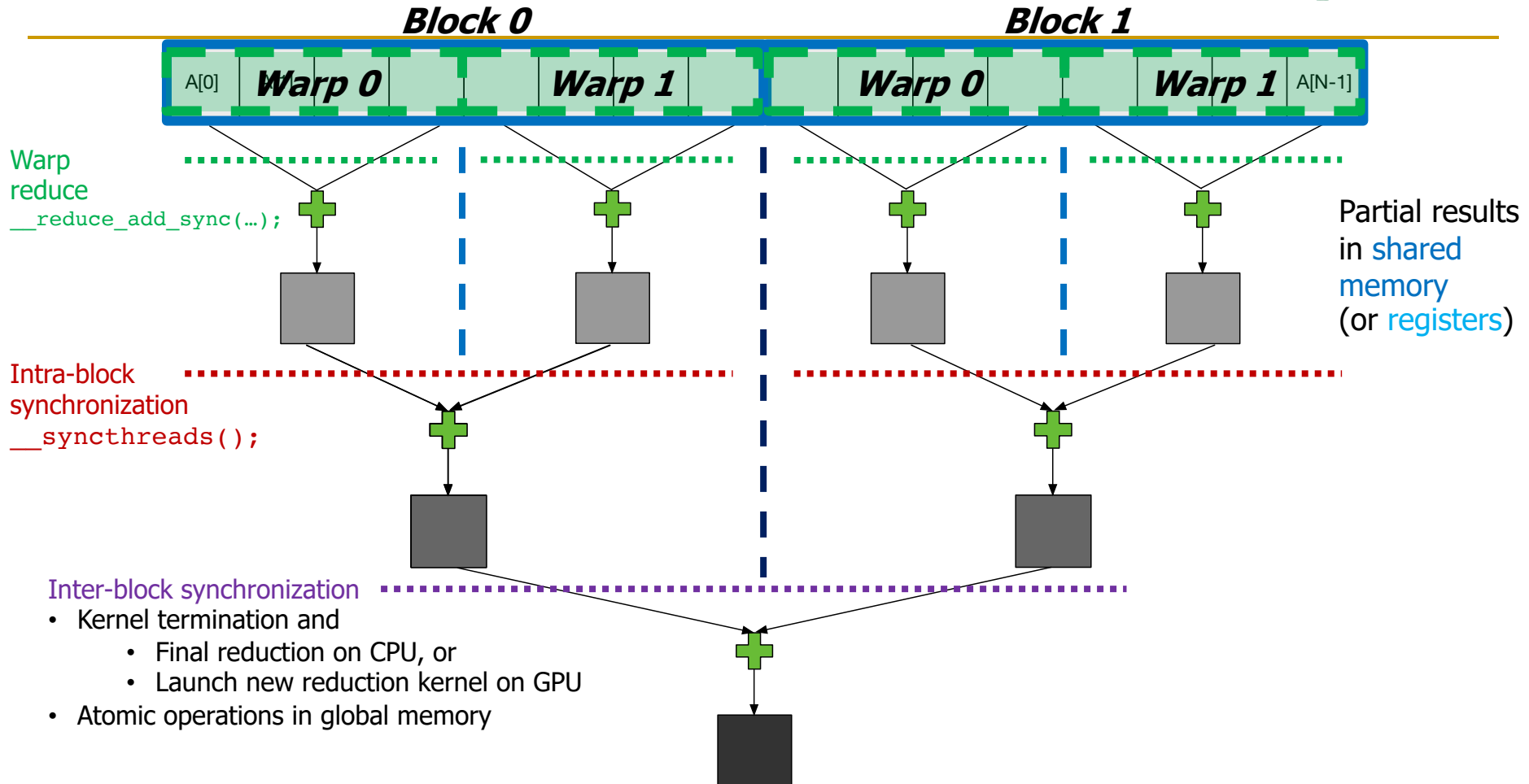
**\_\_reduce\_add\_sync, \_\_reduce\_min\_sync, \_\_reduce\_max\_sync**  
Returns the result of applying an arithmetic add, min, or max reduction operation on the values provided in `value` by each thread named in `mask`.

**\_\_reduce\_and\_sync, \_\_reduce\_or\_sync, \_\_reduce\_xor\_sync**  
Returns the result of applying a logical AND, OR, or XOR reduction operation on the values provided in `value` by each thread named in `mask`.

The `mask` indicates the threads participating in the call. A bit, representing the thread's lane id, must be set for each participating thread to ensure they are properly converged before the intrinsic is executed by the hardware. All non-exited threads named in `mask` must execute the same intrinsic with the same `mask`, or the result is undefined.



# Tree-Based Reduction on GPU (with Warp Reduce)



# Reduction with Warp Shuffle

```
__global__ void reduce_kernel(float* input, float* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ float input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction tree with shuffle instructions
    float sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];

        for(unsigned int stride = WARP_SIZE/2; stride > 0; stride /= 2) {
            sum += __shfl_down_sync(0xffffffff, sum, stride);
        }
    }

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
```

# Reduction with Warp Reduce

```
__global__ void reduce_kernel(int* input, int* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ int input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction with warp reduce instruction
    int sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];

        // Warp reduce intrinsic for cc 8.0 or higher
        sum = __reduce_add_sync(0xffffffff, sum);
    }

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
```

# Atomic Operations (I)

- CUDA provides **atomic instructions** on shared memory and global memory
  - They perform **read-modify-write** operations atomically

- Arithmetic functions

- Add, sub, max, min, exch, inc, dec, CAS

`int atomicAdd(int*, int);`

Return value (old value)

Pointer to shared memory or global memory

Value to add

- Bitwise functions

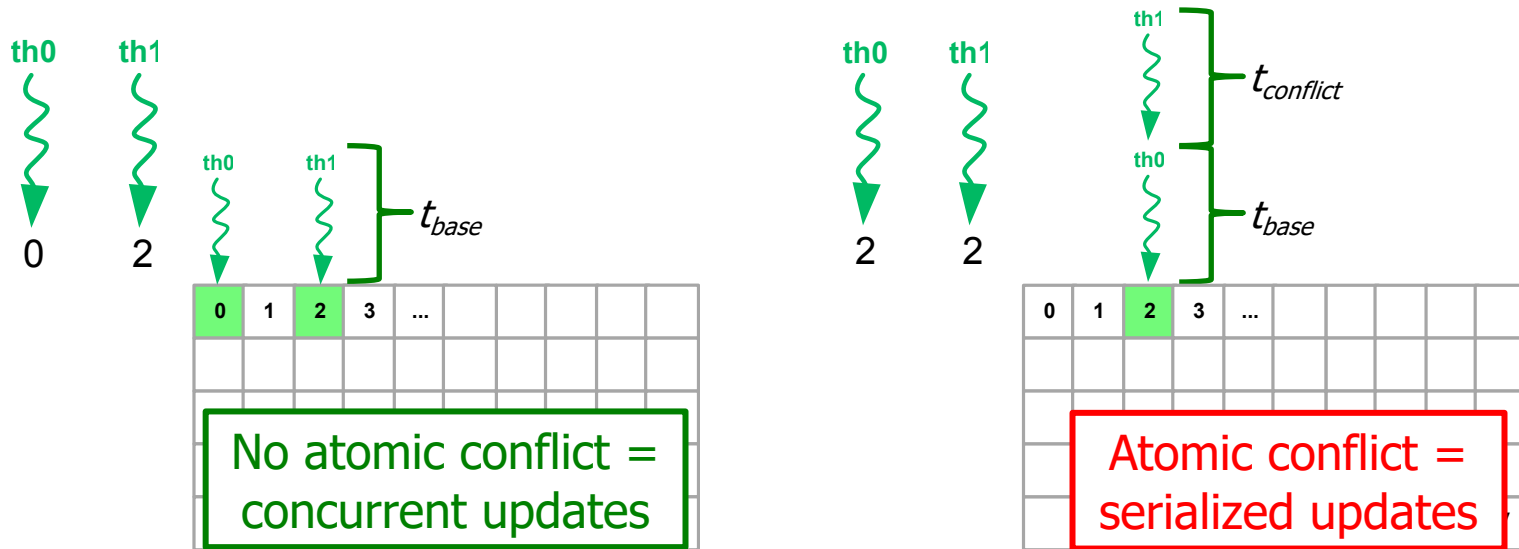
- And, or, xor

- Datatypes: int, uint, ull, float (half, single, double)\*

\* Datatypes for different atomic operations in <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>

# Atomic Operations (II)

- Atomic operations serialize the execution if there are atomic conflicts



# Recall: Uses of Atomic Operations

---

## ■ Computation

- Atomics on an array that will be the output of the kernel
- Example
  - Histogram, reduction

## ■ Synchronization

- Atomics on memory locations that are used for synchronization or coordination
- Example
  - Counters, locks, flags...

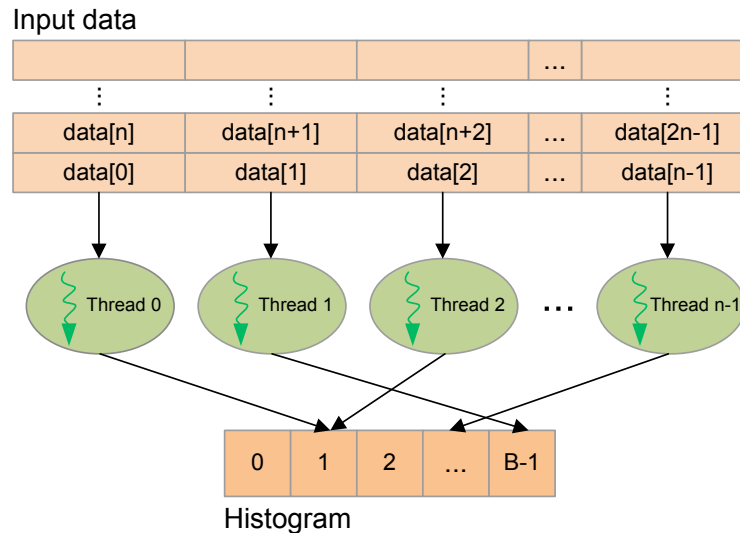
- Use them to prevent **data races** when more than one thread need to update the same memory location

# Image Histogram

- Histograms are widely used in **image processing**
  - Some **computation before voting** in the histogram may be needed

```
For (each pixel i in image I){  
    Pixel = I[i]                // Read pixel  
    Pixel' = Computation(Pixel) // Optional computation  
    Histogram[Pixel']++         // Vote in histogram bin  
}
```

- Parallel threads frequently incur **atomic conflicts** in image histogram computation



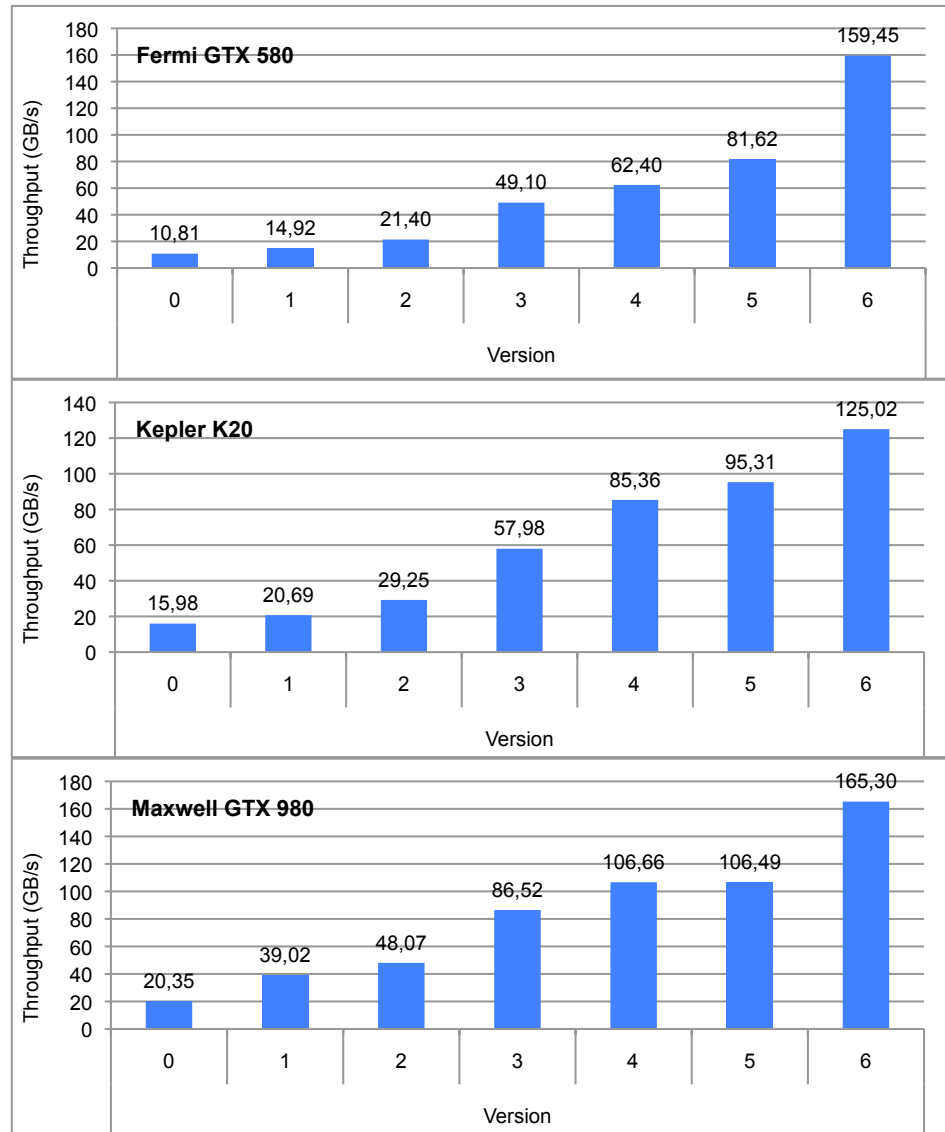
# Optimized Parallel Reduction

---

- 7 versions in CUDA samples: Tree-based reduction in shared memory
  - ❑ Version 0: No whole warps active
  - ❑ Version 1: Contiguous threads, but many bank conflicts
  - ❑ Version 2: No bank conflicts
  - ❑ Version 3: First level of reduction when reading from global memory
  - ❑ Version 4: Warp shuffle or unrolling of final warp
  - ❑ Version 5: Warp shuffle or complete unrolling
  - ❑ Version 6: Multiple elements per thread sequentially



# 7 Versions of Reduction

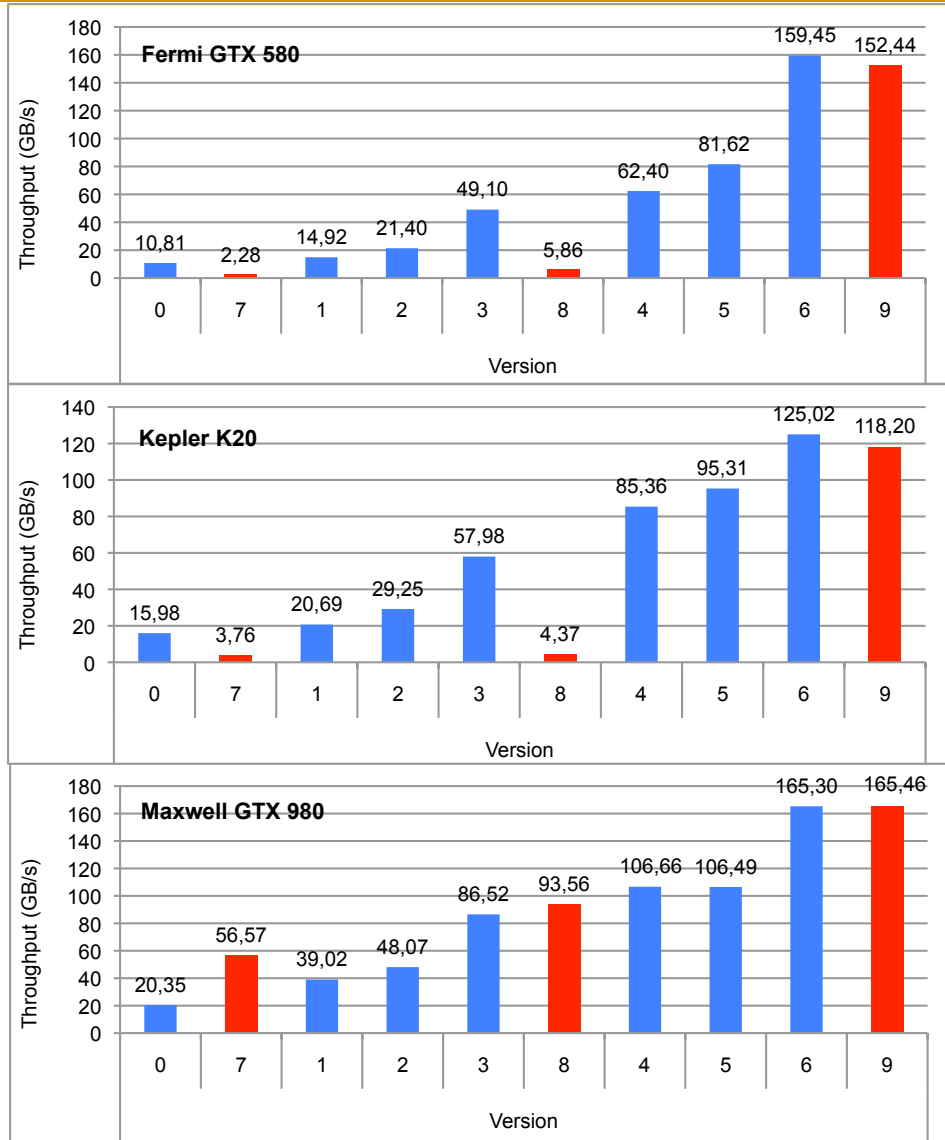


# Reduction with Atomic Operations

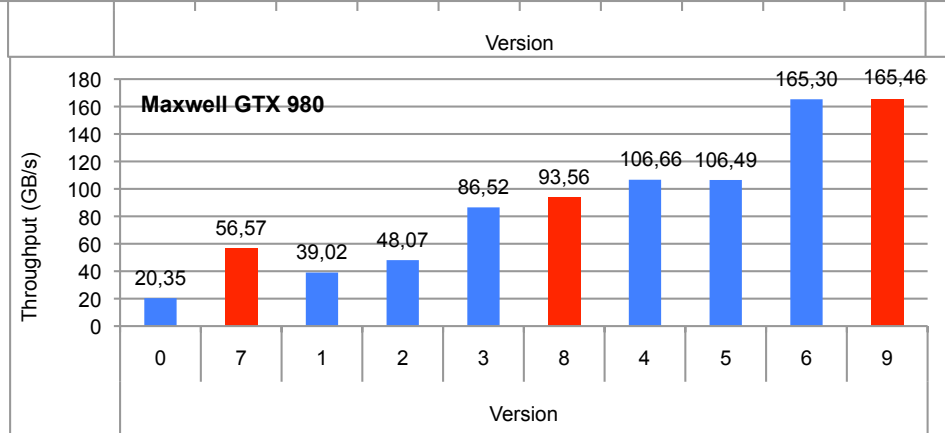
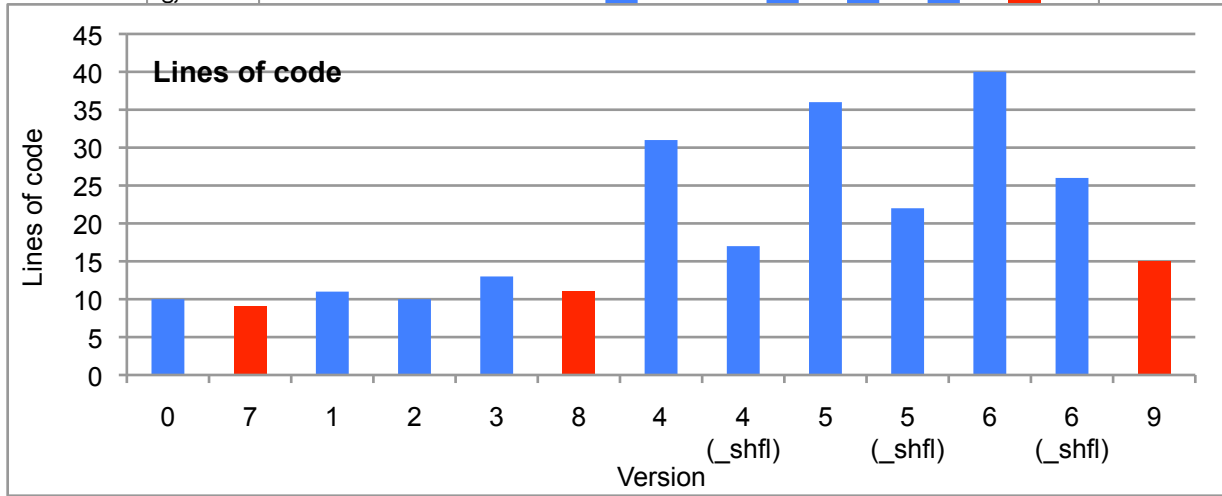
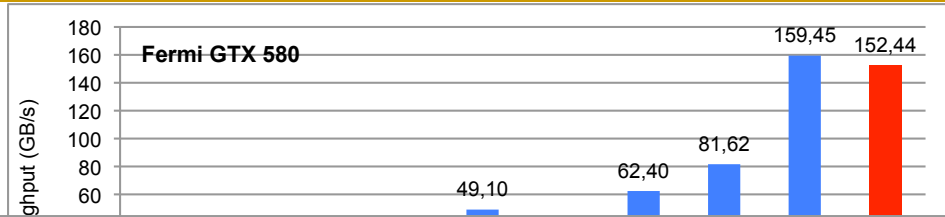
---

- 3 new versions of reduction based on 3 previous versions
  - Version 0: No whole warps active
  - Version 3: First level of reduction when reading from global memory
  - Version 6: Multiple elements per thread sequentially
- New versions 7, 8, and 9
  - Replace the `for` loop (tree-based reduction) with one shared memory atomic operation per thread

# 10 Versions of Reduction

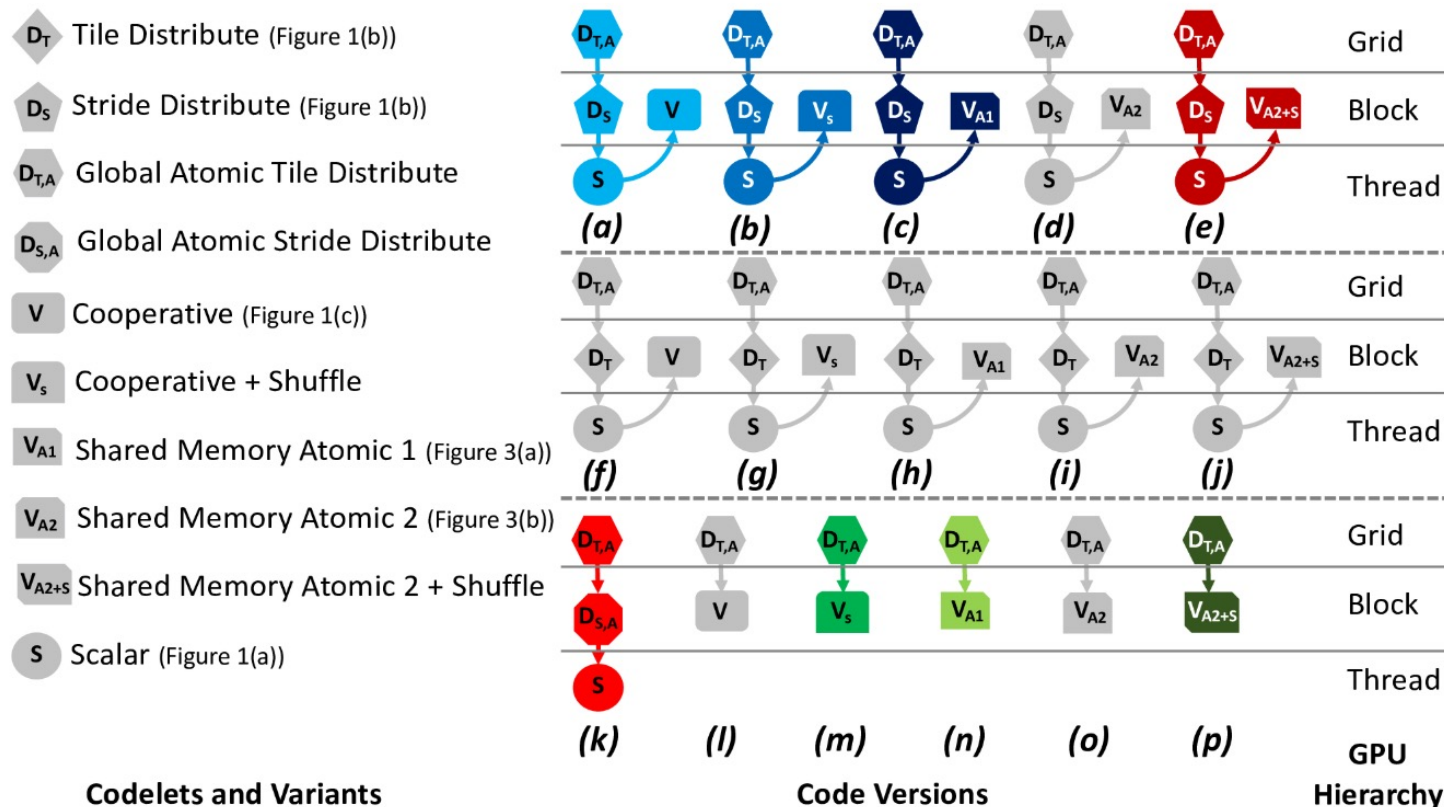


# 10 Versions of Reduction



We save  
lines of code

# Search Space of Parallel Reduction



Over 85 different versions possible!

# Automatic Generation of Parallel Reduction

---

- Simon Garcia De Gonzalo, Sitao Huang, Juan Gomez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu,  
**"Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs"**  
*Proceedings of the International Symposium on Code Generation and Optimization (CGO), Washington, DC, USA, February 2019.*  
[[Slides \(pptx\)](#) ([pdf](#))]

## Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs

Simon Garcia De Gonzalo  
CS and Coordinated Science Lab  
UIUC  
[grcdgnz2@illinois.edu](mailto:grcdgnz2@illinois.edu)

Sitao Huang  
ECE and Coordinated Science Lab  
UIUC  
[shuang91@illinois.edu](mailto:shuang91@illinois.edu)

Juan Gómez-Luna  
Computer Science  
ETH Zurich  
[juang@ethz.ch](mailto:juang@ethz.ch)

Simon Hammond  
Scalable Computer Architecture  
Sandia National Laboratories  
[sdhammo@sandia.gov](mailto:sdhammo@sandia.gov)

Onur Mutlu  
Computer Science  
ETH Zurich  
[omutlu@ethz.ch](mailto:omutlu@ethz.ch)

Wen-mei Hwu  
ECE and Coordinated Science Lab  
UIUC  
[w-hwu@illinois.edu](mailto:w-hwu@illinois.edu)

# Parallel Reduction with Tensor Cores

---

- Reduction can be expressed as a dot product operation and get accelerated by GPU tensor core units
  - $\text{sum} = A_0 * B_0 + A_1 * B_1 + \dots + A_{N-1} * B_{N-1}$
  - With all  $B_i = 1$ , the result will be the sum of array A

## Accelerating Reduction and Scan Using Tensor Core Units

Abdul Dakkak, Cheng Li

University of Illinois Urbana-Champaign  
Urbana, Illinois  
{dakkak,cli99}@illinois.edu

Isaac Gelado

NVIDIA Corporation  
Santa Clara, California  
igelado@nvidia.com

Jinjun Xiong

IBM T. J. Watson Research Center  
Yorktown Heights, New York  
jinjun@us.ibm.com

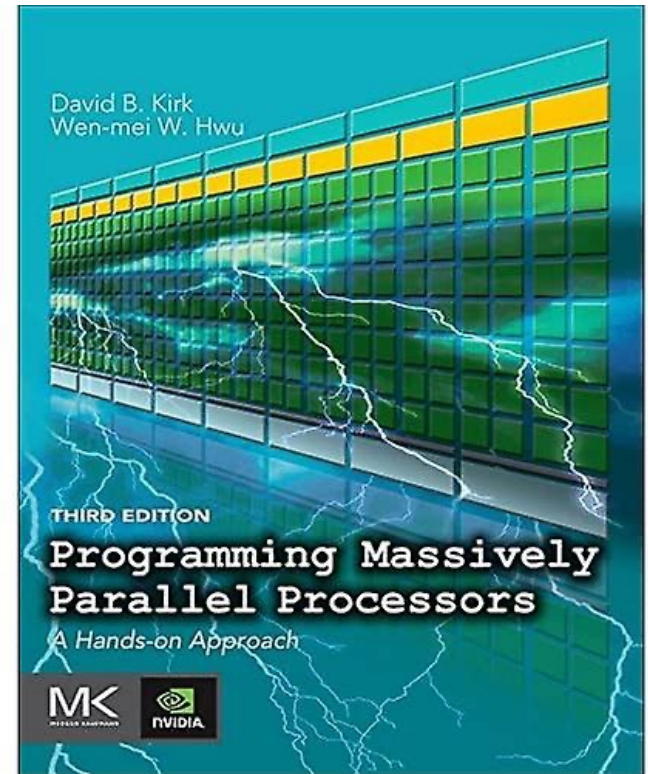
Wen-mei Hwu

University of Illinois Urbana-Champaign  
Urbana, Illinois  
w-hwu@illinois.edu

# Recommended Readings (I)

---

- Hwu and Kirk, “**Programming Massively Parallel Processors,**” Third Edition, 2017
  - Chapter 5: Performance considerations
  - Chapter 9 - Parallel patterns — parallel histogram computation:  
An introduction to atomic operations and privatization

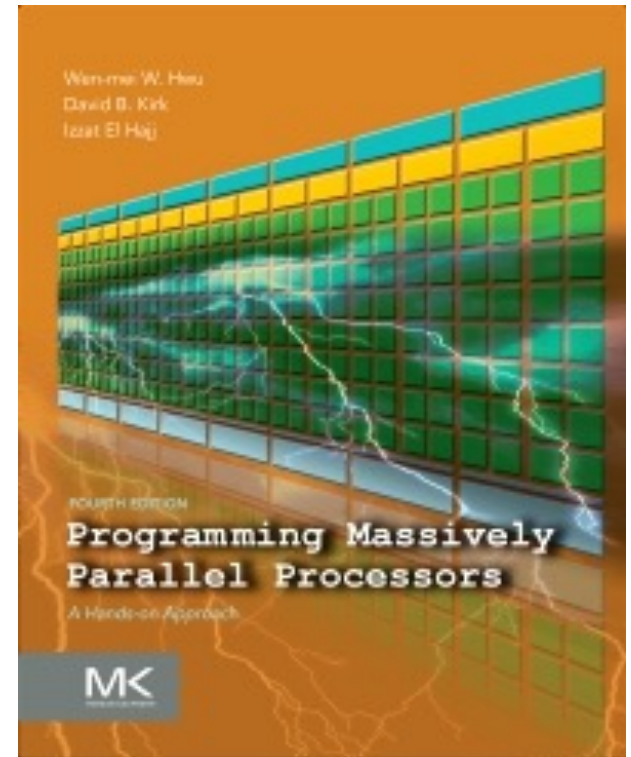




# Recommended Readings (II)

---

- Hwu and Kirk and El Hajj, “[Programming Massively Parallel Processors](#),” Fourth Edition, 2022
  - Chapter 6 - Performance considerations
  - Chapter 10 - Reduction: And minimizing divergence



# P&S Heterogeneous Systems

## Parallel Patterns: Reduction

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2022

7 November 2022